

AFRL-IF-RS-TR-2002-289 Vol 1 (of 2)
Final Technical Report
October 2002



ADVANCED SECURITY PROXY TECHNOLOGY FOR HIGH CONFIDENCE NETWORKS: ADVANCED SECURITY PROXIES

Trusted Information Systems Network Associates, Inc.

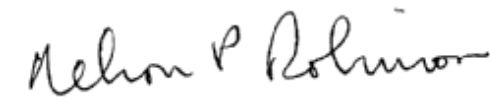
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2002-289 Vol 1 (of 2) has been reviewed and is approved for publication.

APPROVED:



NELSON P. ROBINSON
Project Engineer

FOR THE DIRECTOR:



WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE October 2002	3. REPORT TYPE AND DATES COVERED Final Aug 97 – Jun 01	
4. TITLE AND SUBTITLE ADVANCED SECURITY PROXY TECHNOLOGY FOR HIGH CONFIDENCE NETWORKS: ADVANCED SECURITY PROXIES			5. FUNDING NUMBERS C - F30602-97-C-0336 PE - 33401G PR - NSA1 TA - 00 WU - 01	
6. AUTHOR(S) Roger Knobbe, Andrew Purtell, and Stephen Schwab				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Trusted Information Systems Network Associates, Inc. 3416 South Sepulveda Blvd Suite 700 Los Angeles California 90045			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) NSA / R222 R & E 9800 Savage Road., Suite 6516 Ft. George Meade, MD 20775-6516			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2002-289 Vol 1 (of 2)	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Nelson P. Robinson/IFGB/(315) 330-4110/ Nelson.Robinson@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) This effort addresses the lack of delivery guarantees for encrypted messages that transit through firewall security devices bridging high data rate networks. Firewalls that can easily and dependably handle these high data rates are the main goal of this effort. Generally, current firewall technology is either high-performance or high-security. Defined here is an architecture for decomposing network protocols into the high-security, authentication/access control portion, and the high-performance, high data rate portion. This architecture, used at the transport layer of the OSI-RM, develops the needed firewall proxy software, with the focus on IP over ATM networks and SSL protocol suites. A prototype firewall was developed here that sustains traffic at rates of up to 540 MBS over a Gigabit Ethernet. This was demonstrated in Aug 01 at a DARPA PI meeting.				
14. SUBJECT TERMS Proxy, Gigabit Ethernet, GigaETH, Scout™, Asynchronous Transfer Mode, ATM, Firewall, Hardware Acceleration, IP Cut, Netperf, NETTAP, Open System, Interconnect-Reference Model, OSI-RM				15. NUMBER OF PAGES 35
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

1	Background	1
1.1	ASP Project Background	1
2	Technical Work Accomplished	2
2.1	As-Built ASP Architecture	2
2.1.1	TCP Splicing in a Nutshell	3
2.1.2	NETTAP	4
2.1.3	IPCUT	5
2.2	Proxies	7
2.2.1	HTTP	7
2.2.2	TELNET	8
2.2.3	NETPERF	9
2.2.4	FTP	10
3	Research Results	12
3.1	Hotspots	12
3.2	Performance Graphs and Discussion	14
3.2.1	Fully Proxied	14
3.2.2	Cut-Through	16
3.2.3	Comparative Performance	18
3.3	Patents Applied	19
3.3.1	Shared Congestion State	19
3.3.2	Passive Latency Measurement	20
3.4	Scout Strengths vs. Scout Weaknesses	22
3.4.1	Shoulders of Giants	22
3.4.2	Useful “Path” Abstraction	22
3.4.3	Flexible but Immature Configuration Mechanism	22
3.4.4	Problematic Integration of “round-trip” Routers	23
4	Nature of Problems	23
4.1	Debugging Issues	23
4.2	Device Driver Issues	24
4.3	Workload Characterization Issues	24
4.4	Community “Critical Mass”	25
5	Details of Technical Work	25
5.1	HE622 Driver Details	25
5.2	ACENIC Driver Details	26
5.2.1	Compatibility	26
5.2.2	Chipset-Specific Settings	26
5.2.3	Driver Operation	27
5.2.4	Porting Issues	27
5.3	Hardware Acceleration Options	28
	References	30

Figures

Figure 1 ASP Proxy Architecture.....	3
Figure 2 - Security/Performance Continuum.....	4
Figure 3 NETTAP Snoop Architecture	5
Figure 4 - IPCUT Architecture.....	6
Figure 5 HTTPF Proxy Involvement.....	8
Figure 6 fiNetperf Proxy Involvement	10
Figure 7 FTP Proxy “GET” Involvement.....	12
Figure 8 Processing Time Histogram	13
Figure 9 Derived Processing Histogram.....	13
Figure 10- Derived Cumulative Frequency Distribution.....	14
Figure 11- Steady-state throughput for the full proxy case.	14
Figure 12 - Device-level timing distribution for the fully proxied case.	15
Figure 13 - Timing distribution for combined Ethernet and IP processing for the fully proxied case.....	15
Figure 14 - TCP-level timing distribution for the fully proxied case.....	16
Figure 15 - Steady-state throughput for the cut-through case.	16
Figure 16 - Device-level timing distribution for the cut-through case.	17
Figure 17 - Timing distribution for combined Ethernet and IP processing for the fully proxied case.....	17
Figure 18 - Timing distribution for the IPCUT fast path for the cut-through case.....	18
Figure 19 - Comparative Performance	19
Figure 20 - Shared CCB State	20
Figure 21 - Passive Latency Measurement Device	21
Figure 22 – NAILabs Firewall Performance Testbed	25
Figure 23 - Hardware Assisted NAT	28
Figure 24 - Hardware Assisted IPCUT	29

1 Background

1.1 ASP Project Background

The original goal of the Advanced Security Proxies (ASP) project was to develop high-speed network firewall technology that combines proxy-based firewall mechanisms with high-speed network gateways. The resulting combination is a new firewall proxy platform that provides the strong security of proxies for high-speed networks that current firewall technology did not address in a scalable manner. Rather than sacrificing speed for security, our approach uses flexible techniques that provide for optimal or near-optimal tradeoffs between performance, security, and functionality.

We have developed technology that can control Internet Protocol (IP) traffic carried by high-speed networks (multi-hundred-megabit per second or faster). We have implemented low-level protocol handling techniques that are applicable to any high-speed switched media. The techniques include mechanisms for dynamic “as-needed” re-assembly of lower-level network protocol data units (PDUs) to higher-level PDUs, where the assembly and security processing is performed by interplay between multiple tiers of protocol handling. The purpose of these techniques is to support the functionality of existing firewall proxies, but on a new platform that is oriented toward high-speed networks.

The focus on proxy support is driven by the goal to support the strongest firewall security mechanisms, those that examine application protocol data and base security decisions on content. This is precisely what firewall proxies do*. Therefore, our new ASP firewall platform supports proxies in a manner that is as close as possible to the traditional manner, while eliminating current firewalls’ architectural barriers to high-speed processing. The elimination of these barriers is based on a few principals fundamental to ASP:

- Process application-level data streams for strongest security functionality.
- Perform application-level processing only as needed.
- Data re-assembled as needed, and only as much as needed.
- Multiple tiers of reassembly, including conventional packet filtering where that is sufficient.
- Ability to adaptively switch between tiers as conditions change.

In our prototypes, we demonstrate how to use techniques that meet these principles, to support some currently important proxied protocols. In the following, we use the examples of a “client-protecting” firewall configuration and a “server-protecting” firewall configuration. The “client-protecting” configuration is typified by being “nearer” the client than the server, protecting the client’s network from hostile or unintended network activity from the outside world. The “server-protecting” configuration is typified by being nearer the server, likewise protecting the server’s network from hostile or unintended traffic. Note that it is increasingly common for

*Stateful packet inspectors (SPIs) are another type of firewall component that also can examine application protocol data content. SPIs originated as packet filtering technology, but are becoming functionally similar to proxies. The other distinction between SPIs and proxies is that proxy software (usually user space software) tends to be more separated from the supporting protocol handling (usually a kernel TCP/IP stack). We support the traditional distinction between proxy code and protocol code. Nevertheless, any application-protocol-analysis function should be supported by our ASP platform, whether the function is currently implemented in the context of a proxy or a SPI or both.

transactions to traverse both client-protecting and server-protecting firewalls, although the “opposite-side” firewall is intentionally transparent to the near-side firewall.

2 Technical Work Accomplished

2.1 As-Built ASP Architecture

ASP is a Scout¹-based, hybrid proxy/packet-filtering firewall platform. The Scout operating system supports flexible protocol development and experimentation, and has interesting features for on-host packet routing adaptability. During the early design phase, Scout path and router concepts provided a good vocabulary for describing the ASP adaptive reassembly architecture. Coincident work² indicated that Scout was not only well suited for “adaptively switching between tiers of reassembly”, it had already demonstrated the facility.

One key feature of the Scout OS, fundamental to its capability for adaptable reassembly, is its message de-multiplexing functionality. For our purposes, this de-multiplex phase of message processing allows all packets belonging to a given flow to be rapidly routed through the fastest possible path through the firewall. This de-multiplex machinery is centralized and flexible, allowing flows to be redirected programmatically.

ASP is a “hybrid” platform in its ability to dynamically shift client-to-server network connections from proxied state to packet-forwarded state and back again, programmatically, based on security policy and proxy direction. Reference examples have been written to demonstrate four basic forms of packet processing activity: static TCP-based stream forwarding, static IP-based packet forwarding, dynamic IP packet forwarding based on one-time-use proxy decisions, and dynamic TCP-based stream forwarding based on one-time-use proxy decisions. In addition, the dynamic TCP-based stream forwarding proxies have facilities for either inline stream inspection or offline stream inspection, allowing implementation of security policies based on data location as well as data content.

Static TCP-based stream forwarding and static IP-based packet forwarding are well known and widely available in commercial firewall products³, and will not be further discussed in this paper. Dynamic IP and TCP forwarding, however, are fairly unique and are the features that differentiate ASP from traditional COTS firewalls.

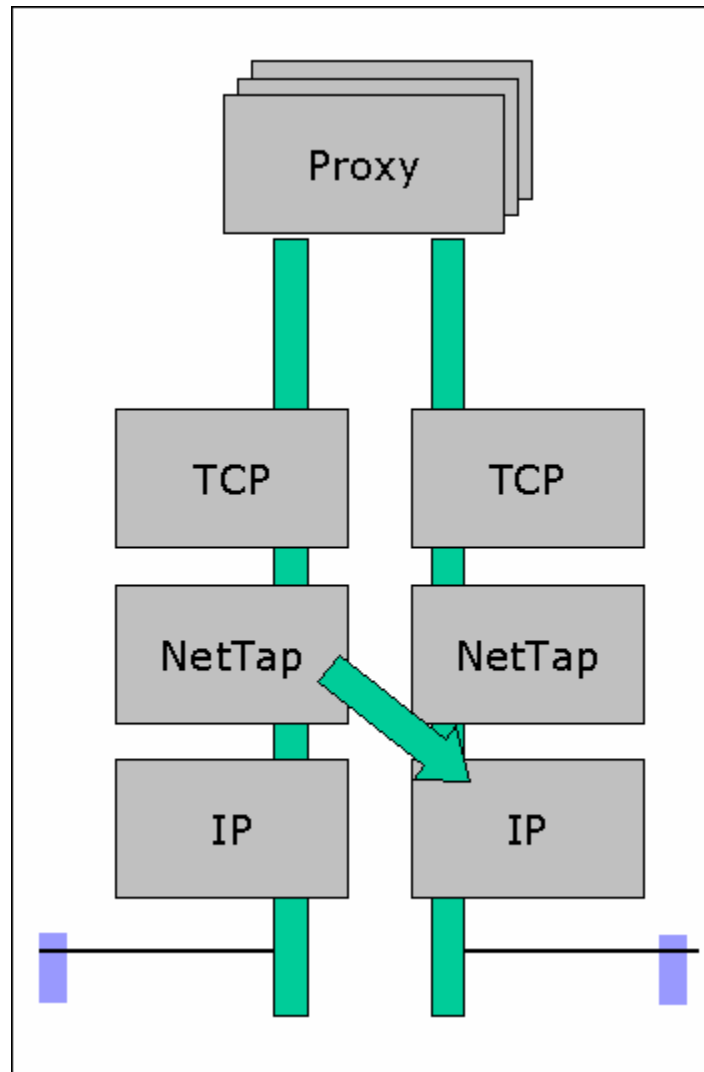


Figure 1 ASP Proxy Architecture

2.1.1 TCP Splicing in a Nutshell

The core of ASP functionality is embodied by the “TCP Splicing” concept. The ASP firewall seeks to marry proxy-style security decisions and router-speed performance, compromising performance only when security decisions must be made. Traditional “proxying” firewalls conduct their work using two pairs of connections: the client connects to the proxy, and the proxy in turn connects to the server. The proxy application actively interprets and copies commands and data between the client- and server-side connections. This arrangement yields high security, as the only data that enters the protected network is data that has been vetted by the proxy.

The traditional alternative to a proxying firewall is a “packet filtering” firewall. Such firewalls are configured with rules to allow certain classes of traffic to pass unimpeded through the firewall. These classes are typically described with rules like: “Forward packets originating from any machine on the protected network onto the unprotected network” and “Forward packets originating from either the FTP-DATA port or the HTTP port of machines on the unprotected network onto the protected network”. These rules are installed into the firewall at configuration time and are static. This arrangement yields high performance, as the packet forwarding

decisions can be made on a frame-by-frame basis and can in general be performed at or near the device driver level.

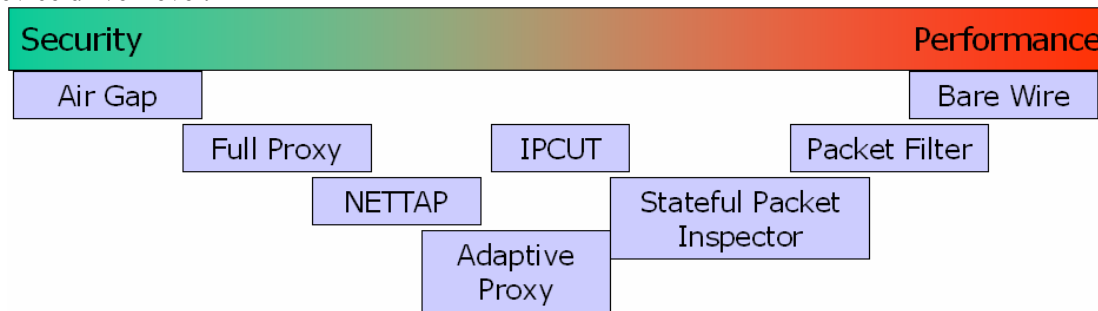


Figure 2 - Security/Performance Continuum

If you consider these two alternatives as being at the opposite ends of a scale, modern firewalls tend to fall somewhere closer to the middle. “Adaptive” proxies establish short-lived packet filtering rules under control of an application-level proxy, improving performance by decreasing the amount of work they must perform on lower-risk traffic. “Stateful” packet filtering firewalls perform rudimentary protocol state following and packet inspection, improving security by increasing the processing on higher-risk traffic.

ASP proxies split this gap once again, following TCP state, yet allowing “trusted” or low-risk traffic to transit the firewall at near-device-driver levels. ASP proxies enjoy the added flexibility of shifting the inspection strategy in response to an updated threat assessment. Inspection may be either *Passive* or *Gated*. Passive inspection describes the case where all or an arbitrary subset of packets transiting a connection may be simultaneously delivered up to the proxy. Gated Inspection describes the case where packets are submitted to the proxy for approval before any packets gain admittance to the network. The proxy may switch between these modes at any time.

Proxy connections requiring security decisions can be grouped into two categories: those which “promote” the security level on an existing pair (client – firewall, firewall – server) of connections, and those in which the security level of an anticipated connection pair can be established before this new connection has taken place. ASP handles these cases using NETTAP and IPCUT mechanisms, respectively.

2.1.2 NETTAP

A “server-protecting” HTTP proxy typifies the NETTAP class of protocols. Consider a security policy in which a web server resides on a protected network behind a firewall. The firewall defends the server from external attacks while still providing access to approved content by trusted clients. This firewall might employ a screening HTTP proxy that examines client credentials and validates HTTP GET requests before passing them along to the web server, but has little or no interest in examining the responses to these requests.

ASP employs NETTAP technology to address this security policy by interposing a partial HTTP filter in the processing stream. This filter, HTTPF, performs both the initial connection establishment and termination commonly found in any HTTP proxy. In addition, once the client credential and URL validation requirements are satisfied, HTTPF sends a SPLICE control messages to NETTAP to initialize the TCP splicing mechanism to edit and forward packets with no further intervention from the HTTPF proxy. In effect, packets recognized as being “on” the spliced TCP connection are forwarded at the same rate as an IP router. The editing performed

upon the packet entails mapping the sequence numbers and ports/addresses to match the expectations of the respective hosts.

ASP supports security policies that require periodic re-examination of the data stream in the response by providing SNOOP technology (Figure 3). SNOOP allows HTTPF to continue to receive the reassembled TCP stream, “tapped” off the spliced TCP connection. While NETTAP prevents HTTPF and the TCP stack on the ASP firewall from actively participating in transport control (i.e. congestion management, flow control, and dropped packet detection), those management messages generated by the client and server are still forwarded and acted upon by the ASP firewall.

A third type of security configuration, INSPECT, is supported as well. The INSPECT model, in which HTTPF and NETTAP collaborate in delivering traffic, allows HTTPF to gate traffic as it passes through a NETTAP store-and-forward queue. ASP does not specify the policy for this gating mechanism, although one might envision bandwidth-limiting policies based on URL, or an adaptive content scanning mechanism tied to external intrusion detection devices. The interaction of this “mute” store-and-forward queue on TCP’s congestion management and flow control has not been fully investigated.

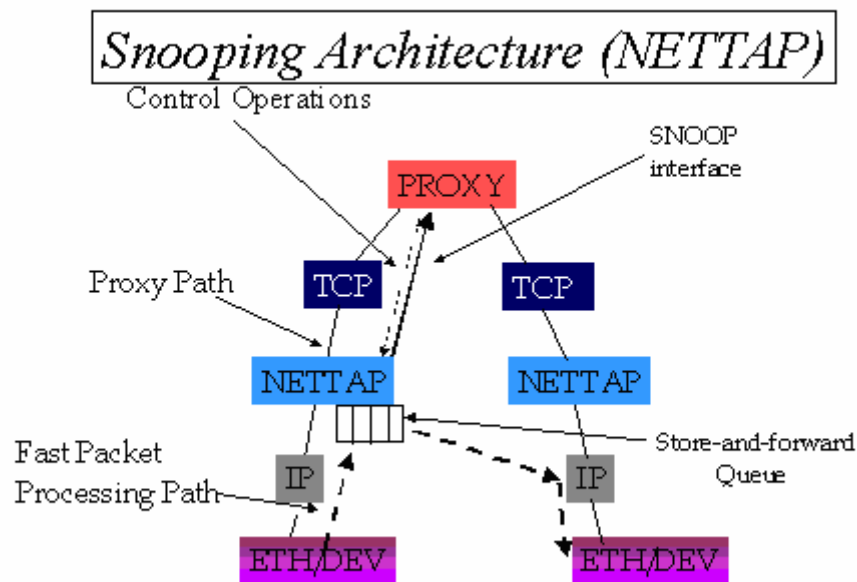


Figure 3 NETTAP Snoop Architecture

2.1.3 IPCUT

The FTP proxy typifies the IPCUT class of proxies. FTP (File Transfer Protocol) transactions utilize two TCP connections – a command connection, over which control and response messages are exchanged between the client and the server, and a data connection, over which bulk data (mostly files and directory listings) is transferred. The FTP client initiates the command

connection to the server, and traditionally[†] the server initiates data connections back to the client, to a client-specified IP address and TCP port.

We have coined the phrase “Bifurcated Command/Data Protocols”, or BCDP’s, to describe these dual-connection protocols. By contrast, we call the single-connection equivalent an ICDP, or Interleaved Command/Data Protocol. As is typically the case in a bifurcated command/data transfer, proxying the command connection is an incomplete solution; knowledge of the protocol is required to parse out connection parameter information and prepare the firewall to securely accept subsequent associated connections. IPCUT provides the mechanism for accepting these subsequent connections with a high degree of confidence in the identity of the initiator and recipient.

IPCUT adds a concept of a “wildcarded passive open”, in which four components of the five-tuple required for a TCP connection (source and destination IP address, destination TCP port, protocol ID) must be matched before the “accept” is completed. The datagram that satisfies the wildcard listen is used to fill in the final component of the five-tuple (the source TCP port), the wildcard becomes “nailed-down” and a fully active open connection.

Consequently, an IPCUTproxy improves firewall security over traditional high-speed firewalls, which rely on protocol specifics (e.g. knowledge of pre-defined TCP source port for FTP data connections) and allocate permanent pass-through rules for datagrams originating from those TCP ports. Because these permanent pass-through rules are stateless and necessarily well known, a common attack crafts a packet that originates from one of these well-known source ports in order to pass unmolested through the firewall.

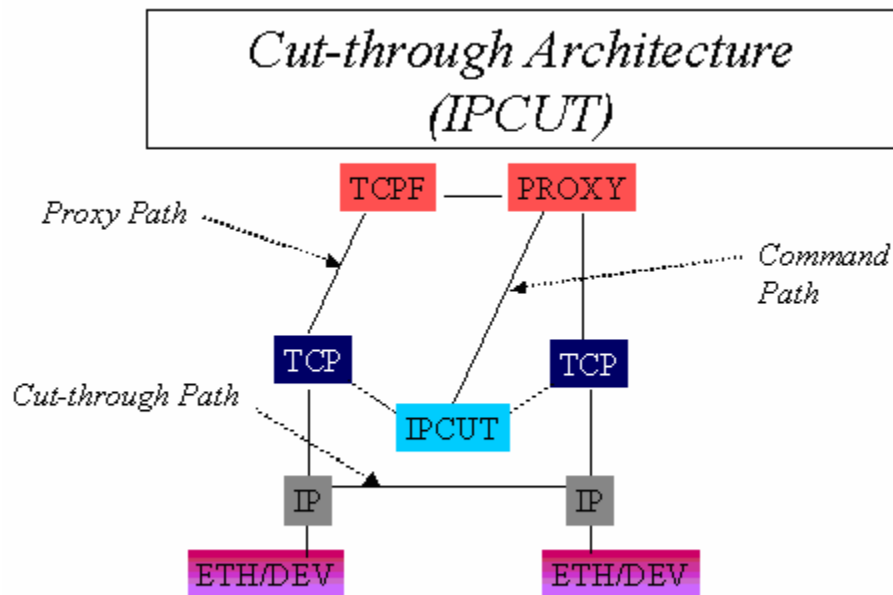


Figure 4 - IPCUT Architecture

[†] The most recent RFCs describing the File Transfer Protocol describe “Passive” FTP data transfers, in which the client initiates both the command and the data connections; the parameters for the data connection are returned in the response to a “PASV” command.

2.2 Proxies

For any given byte in-flight on the Internet, chances are better than 70% that byte either belongs to an HTTP or an FTP-data connection⁴. For any given packet, chances are nearly 50/50 that packet belongs to an HTTP response or an FTP-data connection. Other protocols such as SMTP (mail) and proprietary streaming media (e.g. RealAudio) are completely overshadowed by bulk data and streaming media transfers moving through HTTP and FTP connections. For this reason, we focused our attention on these ubiquitous data transfer protocols.

2.2.1 HTTP

The Hypertext Transfer Protocol (HTTP) delivers Hypertext Markup Language (HTML) pages across the World Wide Web. HTTP Servers, typically running on commodity operating systems, provide data and services via HTTP-based requests and responses.

2.2.1.1 HTTP Protocol

The HTTP standard⁵ defines a simple GET/POST syntax for client-initiated requests, and follows the MIME standard⁶ for encapsulating and describing responses. Recent analysis has shown the typical HTTP request is small (on the order of <500 bytes⁷) and the responses exhibit a heavy-tailed^{8,9} distribution. [I.e. infinite variance] The significance here is that the client-initiated (GET/POST) portion of the transaction contains all the information required to authenticate the client's credentials, validate the access, and initiate the response transfer. For the "server-protecting" firewall configuration, once the firewall has determined the safety of the HTTP request, the response portion of the transfer can proceed unimpeded. For a "client-protecting" firewall configuration, it may be desirable for the firewall to inspect both the client-initiated portion of the transaction (for access to "approved" URLs), and the server response (for safety inspection of mobile code).

2.2.1.2 HTTPF Proxy

The HTTPF proxy implements both variants: client-protecting INSPECT/SNOOP-based proxying, and server-protecting SPLICE-based request validation. The HTTPF proxy can specify which of SPLICE, SPLICE+SNOOP, or SPLICE+SNOOP+INSPECT functionality should be utilized when the cut-through path is created[‡].

[‡] At the moment you set `httpf_use_{join,snoop,inspect}` as appropriate in GDB to get the desired behavior at run time.

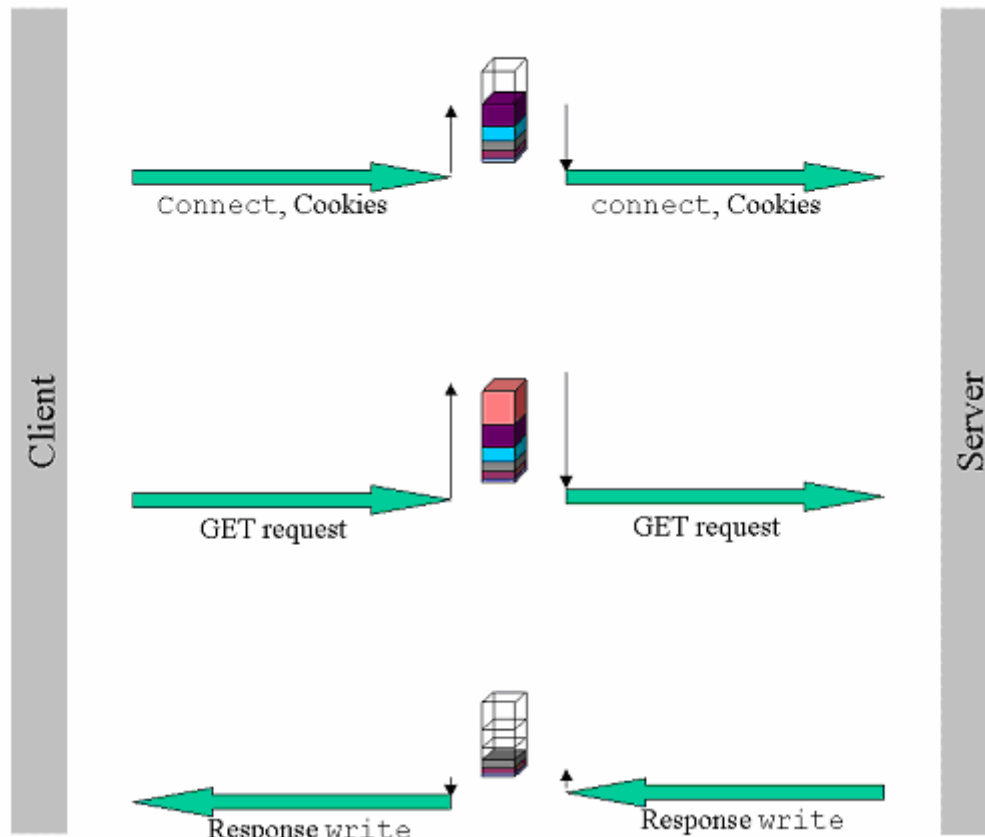


Figure 5 HTTPF Proxy Involvement

2.2.2 TELNET

Telnet¹⁰ is a simple interactive protocol for remote command-line terminal access. While Telnet sessions consume a tiny portion of Internet traffic, the application itself serves a useful purpose for firewall diagnosis, and illustrates a simple Interleaved Command/Data Protocol (ICDP).

2.2.2.1 Telnet Protocol

The Telnet protocol is composed of a single TCP connection, with command messages interleaved within the data stream differentiated from the actual data by a special character sequence. Occurrences of this special character sequence within the data stream are distinguished via a “stuffing” operation, so that a single occurrence in the data stream will actually appear on the wire as two successive sequences.

The Telnet protocol includes a provision for user authentication, however password strings are typically sent in the clear over the wire. Some hosts extend the telnet login exchange with challenge-response or similar secure mechanisms.

2.2.2.2 fiTelnet[§] Proxy

The fiTelnet proxy adds a simple authentication step to the Telnet authentication exchange. This is a proof-of-concept client-protecting firewall configuration. The fiTelnet proxy accepts an outbound client connection, prompts for a secret password, and then establishes the outbound telnet session. Once the outbound telnet session is established, the proxy splices the incoming client-side connection and the outgoing server-side connection together and allows the server's user authentication exchange to take place. Any clients not possessing the secret password are not allowed to utilize the proxy.

The fiTelnet proxy provides a useful comparison point, as it most closely matches the traditional proxy model. All data is reassembled into complete streams, delivered up the inbound TCP connection to the "application level" (although no such distinction between application and kernel exists in Scout), copied to the outbound TCP connection, and re-packetized for transmission. Despite the interleaved command/data characteristics of the protocol, fiTelnet does no interpretation in mid-stream. We use the fiTelnet proxy as a baseline performance metric.

2.2.3 NETPERF

Netperf^{¶1}, while not a widely deployed Internet protocol, is nonetheless a useful and widely accepted diagnostic tool for network performance. Netperf performance statistics are available for a wide range of host hardware, operating systems, and network media, making the Netperf performance statistic the *lingua franca* of network performance analysis.

2.2.3.1 Netperf Protocol

Netperf is a flexible Bifurcated Control/Data Protocol that uses a TCP command channel and can negotiate TCP, UDP, and media-specific (e.g. HIPPI) data channels with various characteristics. The client instructs the server to prepare for a data transfer task with a fixed-format block of data, and the server responds with a similar fixed-format response block containing IP addresses and TCP or UDP port numbers for the subsequent client-initiated data connection. Upon completion of the data transfer task, the server reports on the performance characterization from his perspective (e.g. TCP messages sent), and the client synthesizes the aggregate performance characterization using his own perspective as well

2.2.3.2 fiNetperf Proxy

The fiNetperf proxy (Figure 6) implements a subset of the Netperf protocol, sufficient for a server-protecting firewall configuration, and offering only TCP-based performance testing. The fiNetperf proxy parses and forwards client messages to the server, and parses and forwards server responses to the client. In addition, the fiNetperf proxy interprets server responses with respect to TCP_STREAM connection testing.

The fiNetperf proxy is intended as a client-protecting firewall configuration, however, one in which the firewall is not actually interested in the content of the data connection. Indeed, the firewall would prefer to do as little with the data connection as possible, since this connection is the one that has the "clock running" on it. This configuration is implemented by allowing the proxy to do a "subnet listen", accepting connections from any client on the protected net to the Netperf service port (12865), and parsing the IP destination address from the TCP connection

[§] The "fi" prefix, as in fiNetperf, fiTelnet, fiFTP, etc., is a convention adopted from the Scout system, signifying a "filtering" router, as opposed to a purely functional router. A Scout router provides connectivity and protocol processing sufficient to satisfy end-host requirements. A filtering router allows a restricted subset of the protocol based on some level of protocol-specific access permission.

parameters. The proxy, in turn, establishes an outgoing TCP connection to the Netperf service port on the intended Netperf server.

The fiNetperf proxy establishes an IPCUT connection path upon receipt of the TCP_STREAM_RESPONSE connection setup response message. This IPCUT connection path actually requires the 5 parameters of a TCP connection (Source IP Address & Port, Destination IP Address & Port, and Protocol Number) to be known in advance in order to populate the Scout de-multiplex map. The fiNetperf proxy uses the extended IPCUT “wildcarded passive open” feature to allow a wildcarded connection path, whose parameters may be incompletely specified until known. When the client receives the edited TCP_STREAM response message and initiates the data connection to the server (via, transparently, the firewall), the final parameter (the Source TCP port) required for the Scout de-multiplex machinery is available and completes the connection path specification.

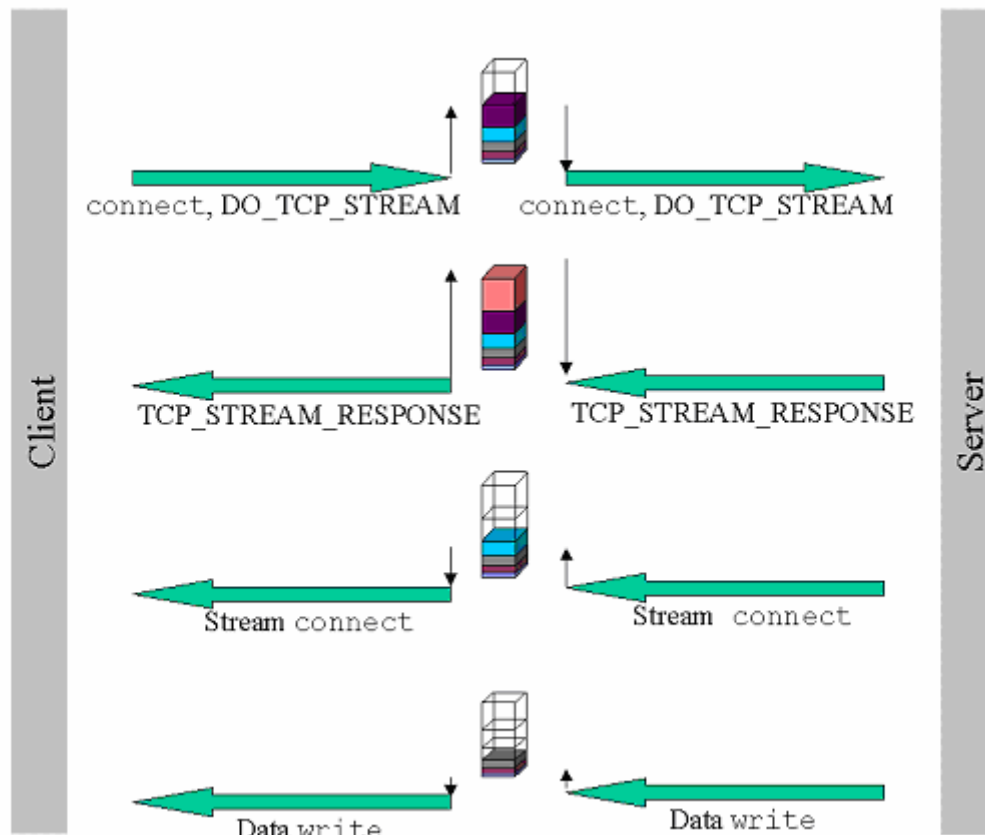


Figure 6 fiNetperf Proxy Involvement

2.2.4 FTP

FTP¹² is a BCDP file transfer protocol. FTP is a much older protocol than HTTP, and was commonplace prior to WWW clients and servers being widely deployed. As such, there are still large bodies of data available on legacy FTP-based servers around the world.

2.2.4.1 FTP Protocol

FTP uses TCP connections for both command and data traffic. Command traffic exchanges user credentials and simple data transfer initiation commands. Commands and responses that initiate data transfer include parameters for data connection establishment. Files and “large” responses traverse these data connections, while status-messages and “small” responses are delivered in-line in the command channel. FTP Data connections may be either “active” or “passive”, determined by client activity. In active FTP data connections, the client directs the server to an opened, listening port on the client machine. The server, in turn, initiates a connection to that port in order to carry out the data transfer. In passive FTP data connections, the client indicates a desire to initiate a passive connection using the “PASV” command. The server’s response directs the client to a new open, listening port on the server. The client, then, initiates the data transfer by connecting to the indicated port. The originating host nonetheless carries out data transfer once this connection is established.

2.2.4.2 fiFTP Proxy

The fiFTP proxy (Figure 7) implements a subset of the FTP protocol; sufficient for server-protecting firewall configurations. The proxy parses FTP command messages from the client, and command response messages from the server, and forwards these messages respectively. Data channel negotiation messages (i.e. PORT commands) are interpreted and redirected to a wildcarded port the proxy opened on the firewall. When the server initiates the data connection, the firewall observes the final connection parameter and “nails down” the wildcarded port. From this point on, all packets in the TCP data connection are forwarded from source to destination as soon as they are recognized.

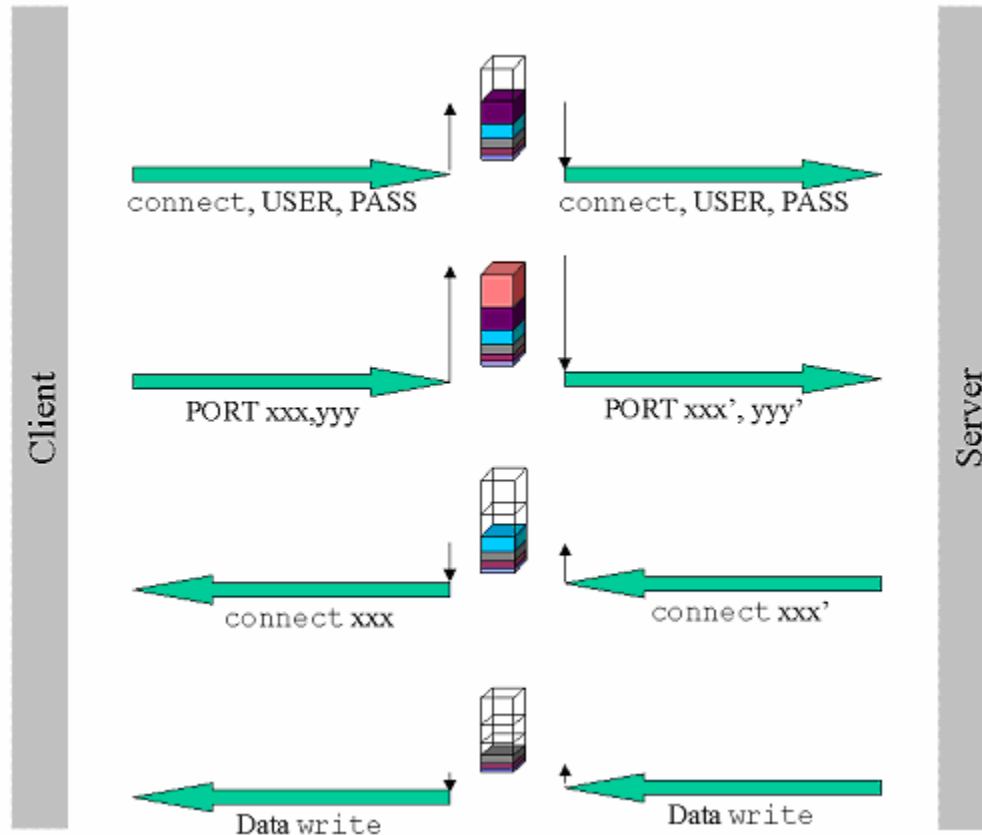


Figure 7 FTP Proxy "GET" Involvement

3 Research Results

3.1 Hotspots

ASP has augmented the Scout operating system with profiling and "micro-benchmarking", message-specific timing statistics for the various phases of protocol processing. As in any system, it is often the case that a few "hot spots" dominate performance characteristics, and optimization efforts are best focused here. Micro-benchmarks have helped us to isolate these hotspots and make performance predictions based on "what-if" models.

ASP code was profiled utilizing the free-running CPU counter registers available in both the Compaq Alpha AXP-series and the Intel Pentium-series microprocessors. Scout "message" structures were augmented with several independent timestamp fields. Each processing layer accounted time spent within that layer for each individual message. As layers complete processing for a given message, the accumulated processing time for that message is added to the appropriate histogram ring-buffer. Later, based on operator intervention, (i.e. a debugger breakpoint) the ring-buffers are examined and a histogram chart is produced. One of these histograms is shown in Figure 8. The numeric information in these histograms is typically reformatted into graphical charts for presentation (see Figure 9 and Figure 10)

NOTE: The histogram instrumentation, while fairly lightweight, is still an intrusive measurement device. For this reason, all raw performance measurement, including throughput measurement, are made with histogram collection disabled. All results below, which group throughput with microbenchmark histograms, reflect multiple collections of benchmark runs.

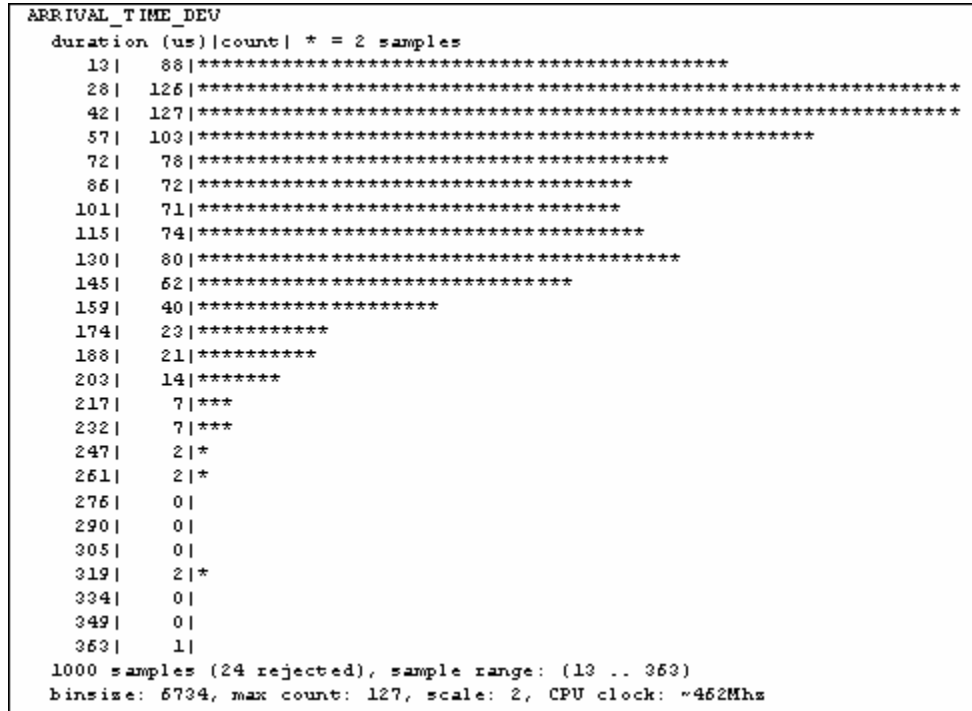


Figure 8 Processing Time Histogram

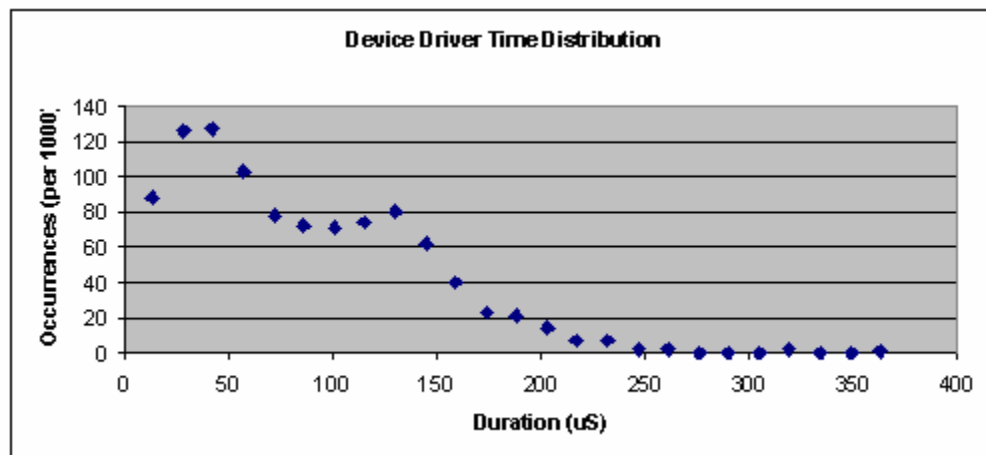


Figure 9 Derived Processing Histogram

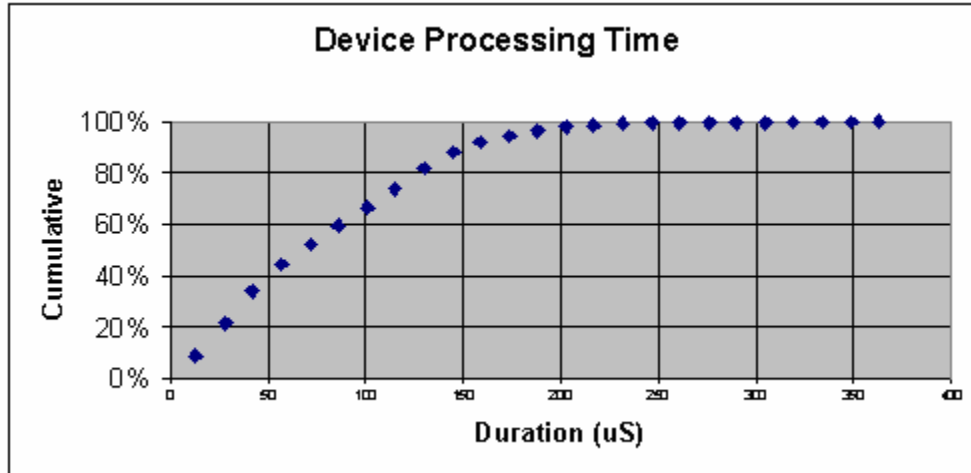


Figure 10- Derived Cumulative Frequency Distribution

3.2 Performance Graphs and Discussion

To demonstrate the performance improvement potential of the ASP architecture, we chose to compare the throughput rates and processing requirements of a simple “fully proxied” TCP connection versus a cut-through BCDP utilizing the IPCUT framework. The fully proxied baseline case uses a non-authenticating version of the fiTelnet proxy, configured to copy data at the application layer from one inbound TCP connection to a corresponding outbound TCP connection, and vice versa. The cut-through IPCUT proxy is the fiNetperf proxy described above.

3.2.1 Fully Proxied

Baseline measurements were performed using the `ttcp`¹³ program. Throughput was measured with the two standard MTU sizes for Gigabit Ethernet networks, 1500 bytes and 9000 bytes. Microbenchmark data was collected for a representative sample of traffic. Tests were performed with Compaq DS10 (Alpha 21264 processors) and 3Com 3c985 Gigabit Ethernet interfaces. Three DS10’s were deployed in the testbed: One each as source, sink, and firewall. Source and sink nodes were running the Linux operating system running version 2.4 kernels and device drivers. The firewall node was running ASP.

Fully proxied network traffic is fairly CPU intensive, as is indicated by the following timing distribution diagrams. Throughput for MTU 9000 packets is roughly twice that for MTU 1500 packets, despite the sixfold increase in size, indicating that there is a non-linear cost associated with processing packets of larger payload.

	message size (bytes)	time (sec)	throughput 10 ⁶ bits/sec
MTU 1500	8192	30	161.44
MTU 9000	8192	30	368.64

Figure 11- Steady-state throughput for the full proxy case.

The composite timing distribution for fully proxied traffic (Figure 12, below) shows the mode at roughly 1160 microseconds total processing time per second. The subsequent timing

distributions show that this time is roughly distributed as half Ethernet+IP processing (Figure 13, below), and half TCP processing (Figure 14).

```

ARRIVAL_TIME_DEV
duration (us)|count| * = 1 samples
<1| 24| *****
96| 0|
193| 9| *****
290| 14| *****
386| 13| *****
483| 16| *****
580| 26| *****
677| 20| *****
773| 27| *****
870| 37| *****
967| 37| *****
1063| 32| *****
1160| 51| *****
1257| 40| *****
1354| 40| *****
1450| 33| *****
1547| 28| *****
1644| 29| *****
1740| 23| *****
1837| 21| *****
1934| 18| *****
2031| 7| *****
2127| 0|
2224| 1| *
2321| 1| *

```

Figure 12 - Device-level timing distribution for the fully proxied case.

```

ARRIVAL_TIME_IP
duration (us)|count| * = 1 samples
<1| 22| *****
39| 2| **
79| 2| **
118| 5| *****
158| 9| *****
198| 47| *****
237| 59| *****
277| 46| *****
317| 62| *****
356| 78| *****
396| 59| *****
435| 34| *****
475| 31| *****
515| 26| *****
554| 21| *****
594| 9| *****
634| 8| *****
673| 7| *****
713| 6| *****
752| 4| *****
822| 3| *****

```

Figure 13 - Timing distribution for combined Ethernet and IP processing for the fully proxied case.

```

ARRIVAL_TIME_TCP
duration (us)|count| * = 1 samples
0| 23| *****
39| 1| *
78| 2| **
117| 5| *****
156| 11| *****
196| 55| *****
235| 53| *****
274| 48| *****
313| 60| *****|
352| 75| *****|
392| 59| *****|
431| 32| *****
470| 30| *****
509| 27| *****
549| 22| *****
588| 8| *****
627| 9| *****
666| 6| *****
705| 7| *****
745| 0|
784| 4| ****
823| 3| ***

```

Figure 14 - TCP-level timing distribution for the fully proxied case.

3.2.2 Cut-Through

ASP improvements to the basic firewall architecture are demonstrated below. The Netperf protocol was used to measure TCP forwarding at the IPCUT level. Machine configuration was identical to the Fully Proxied configuration above.

	Recv Socket bytes	Send Socket bytes	Send Message bytes	Elapsed Time	Throughput 10^6bits/sec
MTU 1500	131070	131070	8192	30.00	229.08
MTU 9000	131070	131070	8192	30.00	549.05

Figure 15 - Steady-state throughput for the cut-through case.

As in the Fully Proxied case, relative throughput between the MTU 9000 case and the MTU 1500 case indicates the cost of processing is tied more strongly to per-message factors rather than per-byte factors. However, the removal of the TCP processing overhead substantially reduces the amount of processing required per message and substantially improves overall throughput. The composite timing distribution (Figure 16) is shifted dramatically with the mode appearing at 36 microseconds and the majority of per-message processing consuming less than 100 microseconds. Indeed, Figure 17 and Figure 18 show that virtually all processing outside the device driver consumes less than 10 microseconds, or 10% of the CPU budget.

```

ARRIVAL_TIME_DEV
duration (us)|count| * = 3 samples
14| 72|*****
25| 150|*****
26| 160|*****
47| 125|*****
58| 133|*****
69| 79|*****
80| 74|*****
91| 57|*****
102| 37|*****
113| 23|*****
124| 25|*****
135| 17|*****
146| 9|***
157| 7|**
168| 5|*
179| 4|*
190| 4|*
200| 3|*
233| 1|
244| 2|
255| 2|
266| 0|
277| 1|

```

Figure 16 - Device-level timing distribution for the cut-through case.

```

ARRIVAL_TIME_IP
duration (us)|count| * = 17 samples
3| 881|*****
7| 25|**
11| 19|*
16| 19|*
20| 2|
24| 2|
28| 2|
32| 4|
37| 2|
41| 9|
45| 5|
49| 4|
53| 7|
58| 3|
62| 1|
66| 0|
70| 0|
74| 0|
79| 1|
83| 0|
87| 1|
91| 0|
95| 1|

```

Figure 17 - Timing distribution for combined Ethernet and IP processing for the fully proxied case.

```

ARRIVAL_TIME_IPCUT
duration (us)|count| * = 19 samples
<1| 989|*****
2| 0|
4| 0|
6| 6|
8| 0|
10| 0|
12| 1|
14| 2|
16| 0|
18| 0|
20| 0|
22| 0|
24| 0|
26| 0|
28| 0|
30| 0|
32| 0|
34| 0|
36| 0|
38| 1|

```

Figure 18 - Timing distribution for the IPCUT fast path for the cut-through case.

3.2.3 Comparative Performance

While protocol-versus-protocol comparisons are somewhat deceptive, it is interesting to note the relative performance characteristics of host-based packet forwarding, direct (bare-wire) throughput, and ASP-based secure packet forwarding. Figure 19 shows a rough spectrum of the kinds of throughput achieved by the following configurations:

- ASP MTU 1500 – Netperf across Scout/ASP firewall running IPCUT-derived proxy with 1500-byte MTU.
- ASP MTU 9000 – Netperf across Scout/ASP firewall running IPCUT-derived proxy with 9000-byte MTU.
- Linux MTU 1500 – Netperf across Linux running as a simple IP-Forwarding router, 1500-byte MTU.
- Linux MTU 9000 – Netperf across Linux running as a simple IP-Forwarding router, 9000-byte MTU.
- Direct MTU 9000 – Netperf across Point-to-Point (crossover cable) connection between source and sink, no firewall, 9000-byte MTU.
- Ttcp mtu 1500 – Ttcp across Scout/ASP firewall running TCPF-derived proxy with 1500-byte MTU.
- Ttcp mtu 9000 – Ttcp across Scout/ASP firewall running TCPF-derived proxy with 9000-byte MTU.

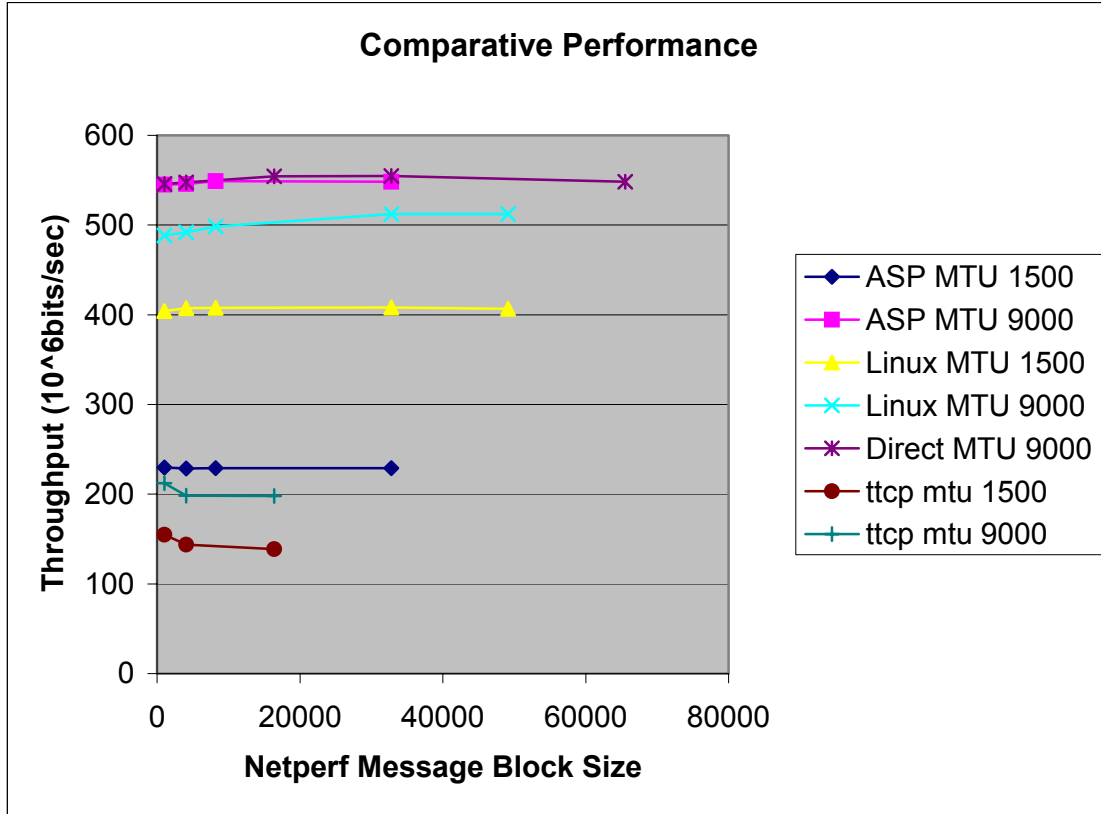


Figure 19 - Comparative Performance

The comparative results indicate that the ASP IPCUT configuration runs at near-bare-wire speeds, 1500-byte MTUs are much less efficient than 9000-byte MTUs, particularly for ASP but even for simple packet forwarding, and, interestingly, that the ASP IPCUT configuration presents better performance than the Linux packet-forwarding router for large MTUs.

3.3 Patents Applied

Our increasing level of expertise with Scout and the modifications to the Scout TCP stack has lead us to investigate other functional areas within advanced networking. Scout is a comfortable platform for network stack experimentation and has a powerful vocabulary for expressing certain types of network concepts. Two patent applications have been filed based on our work and experience with the Scout platform.

3.3.1 Shared Congestion State

Our enhancements to the Scout TCP congestion management facility, originally intended to improve responsiveness to network congestion and collision avoidance, presented us with an opportunity to improve the performance of firewall “clusters”. Traditionally, hosts maintain congestion and window information per network connection. Any new connections recalculate this congestion state information; walking through the same slow-start method and slowly ramping receive windows up until finding the break-even point of window size versus bandwidth availability. For a proxying firewall, this slow startup is quite limiting, as it is quite common for a web browser to establish multiple connections to the same host simultaneously in order to parallelize data fetches and better utilize network bandwidth. In fact, each new connection suffers

the same slow-start delay while ramping up to the full bandwidth available to the connection. For the typical small HTTP fetch, the slow-start delay dominates the transaction time.

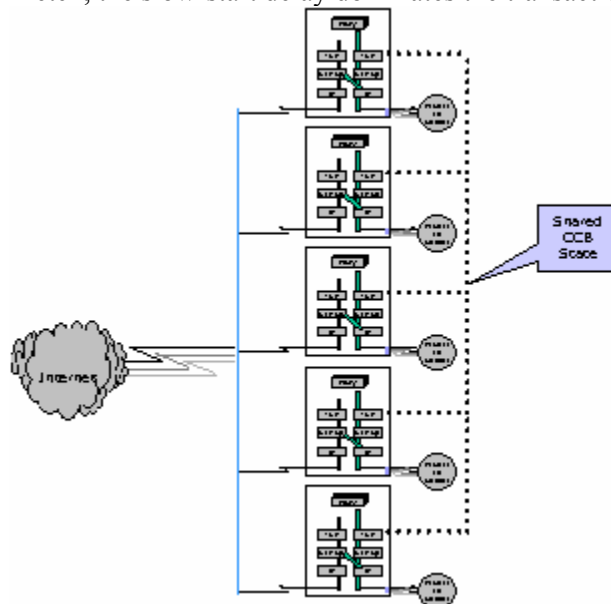


Figure 20 - Shared CCB State

ASP's Scout stack maintains "shared" congestion state, in which it maintains per-host (rather than per-connection) state. The benefit is that new TCP connections to recent hosts will not suffer from slow-start and can begin transferring data with the most recently calculated optimal parameters. This shared congestion state is actively maintained by one single connection, the "master", who periodically updates the other "slave" congestion state copies. Controller responsibilities are rotated in round-robin fashion, when the current round-trip-time sample is complete, from active connection to active connection. This minimizes the risk that a silent connection will allow congestion statistics to stagnate for other active connections to that same host.

The patentable idea is that this shared congestion state is valuable not just for a single firewall machine, but also across multiple constituent firewalls in a firewall cluster. The system described by the patent marshals the shared congestion state information on each firewall host and makes this information available to all the other cooperating firewalls in the system. In the event a firewall chooses to make a connection to a machine outside of his current "working set" of connections, the firewall can ask his peers for reasonable initial congestion state values. The congestion state information for that host includes Round Trip Time estimates (RTT), Path Maximum Transmission Unit (PMTU), and Congestion Window Size (CWIN).

3.3.2 Passive Latency Measurement

Network device performance characterization is generally described by a number of variables: bandwidth, cost, reliability, and latency. Bandwidth is straightforward to measure; Netperf, described above, is a common measurement tool. Cost is easy. Reliability is estimable. Latency is hard.

Hardware-based measurement devices inject easily recognizable data patterns into a device and watch for these patterns to emerge from the device. Latency is calculated by the time difference between the injection and recognition. While these "flag" frames can be injected into bridges,

switches, or hubs to obtain accurate and fair latency measurements, active devices such as routers and firewalls are a different story. It is conceivable for a firewall or router to be configured to forward “flag” frames unimpeded in support of latency measurement, however this is hardly a fair estimate of the true latency experienced by actual traffic.

On the other hand, a measurement tool with the intention of estimating latency in terms of round-trip time could generate realistic traffic through the active device. While this technique is more likely to yield fair results (assuming the device can actually generate “realistic” traffic), the measurement is not directly reducible to the latency of the device under test. Many more measurements must be performed to isolate the various round-trip times in the equation and reduce the number of external delays. At best, the measurement would represent the aggregate latency experienced transiting through the network device by the outbound message plus the return message.

Furthermore, it is not always feasible to inject additional traffic into an existing network configuration; it may be the case that additional measurement traffic will unfairly skew the results (suppose the system is running at or close to capacity and additional traffic will congest the network.) Finally, it is possible that a router or firewall vendor could anticipate measurement traffic streams and grant preferential treatment to any suspected measurement streams. Such systems would falsely represent their performance with respect to real world traffic.

What is needed is a method of passively measuring latency through a network device. We have applied for a patent on a design for such a device, based on Scout. This scheme works by collecting traffic on the input and output legs of the device-under-test and correlating the traffic streams based on the various protocol layers and transformations recognized. As correlations are achieved, the timestamps of the constituent bytes are compared and latency measurements are reported. Since correlations are being attempted at multiple levels, (MAC-level, IP datagram level, TCP level, application level, etc.) multiple simultaneous estimates of latency will be reported. Latency is expected to vary over time, as well, based on changes in internal processing requirements due to data stream and hardware events. We anticipate dynamically updated “Latency Gauges” which indicates a low-water and high-water mark of latency measurements at the various levels of correlation.

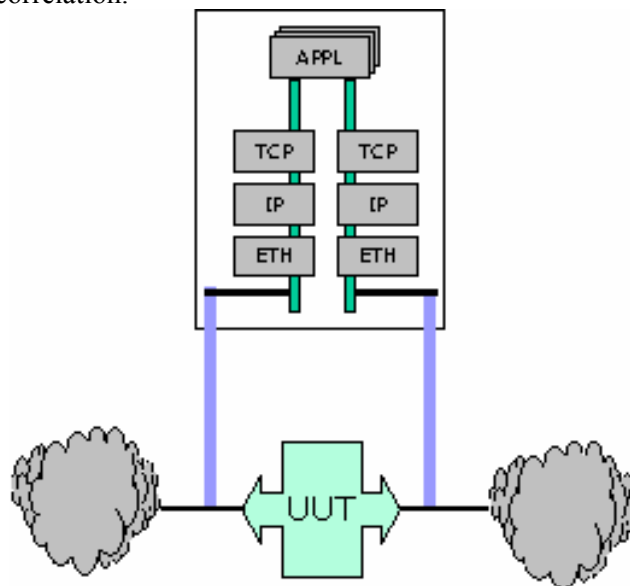


Figure 21 - Passive Latency Measurement Device

3.4 Scout Strengths vs. Scout Weaknesses

One of the secondary benefits of the ASP projects has been the independent evaluation of the Scout platform as a network device deployment tool. To our knowledge, NAILabs is the only research body outside the Arizona/Princeton community to use Scout in a large-scale exercise. We now take this opportunity to critique the Scout platform as a network device RAD (Rapid Application Development) tool.

3.4.1 Shoulders of Giants

The Scout codebase is a powerful, well-designed and well-documented, radical new direction for operating system design. Scout provides a rich library of OS building blocks, including zero-copy memory buffers, efficient hash tables, synchronization primitives and a complete, high-performance TCP/IP networking stack. Without these core components, implementing a high-performance firewall from scratch would be a formidable, if not impossible task. We owe a huge debt of gratitude to the Scout designers and developers for laying the foundations of ASP in Scout.

That being said, the Scout codebase suffers from a lack of contributors. Device driver development has stagnated (with the notable exception of the SILK¹⁴ work and its support for native Linux devices.) Less glorious projects like the user interface and the configuration system, while desperately in need of improvement, take a backseat to trendy or experimental OS improvements. This is not surprising of a common vehicle for networking-related Masters and Doctoral theses. However, growth of the user community is hampered by the lack of such superficial features. For better or worse, the addition of superficial features is directly proportional to the size of the developer community.

The Open Source model has shown that below a certain size of developer base, the burden of maintenance and the concept of ownership remain with the original developers¹⁵. As the Scout system has an academic “founder” constituency, the ownership changes hands every few years as the academic and student contributors move through their careers. We have seen recent renewed activity out of Princeton (Larry Peterson’s new home), so the situation seems to be improving.

3.4.2 Useful “Path” Abstraction

The Path concept, key to the Scout Architecture, is a powerful abstraction of data flow. Paths map directly into the ASP processing model of trusted flows. We like the path, and plan to use this abstraction method for future work, both in the patented ideas presented in Section 3.3 as well as in future DARPA networking research endeavors.

3.4.3 Flexible but Immature Configuration Mechanism

The original designers provided a configuration “Little-Language”¹⁶ for describing the protocol processing elements and their interconnection. This arrangement allows a great deal of flexibility in laying out the networking components. Protocol modules (“routers”) are described with named, typed interfaces in a specification (`.spec`) file. Router configuration is performed at compile time, and the configuration process creates header files and populates a C source file which instantiates each router instance.

While this flexibility is powerful, and the degree of abstraction it provides does allow the entire “Fast Path” concept, flexibility is a two-edged sword. During development, router configuration

specifications (`config.graph` files) are practically static. Moreover, subgraphs of the configuration graph are partially ordered (e.g. TCP is always at least indirectly “above” IP, and anticipates this relative positioning in its code) despite the fact that the configuration language has no means to ensure this relationship and no mechanism for the routers to indicate such dependencies. The configuration flexibility, therefore, is incomplete and often unwarranted.

The increased complexity associated with maintaining the layer of indirection between routers often results in code that is subtly broken with respect to the linkage indirection pointers. To make matters worse, the relatively static nature of a developer’s working `config.graph` file implies that flaws in the linkage code between routers aren’t revealed until the `config.graph` file changes substantially, typically when a *different* developer picks up previously “working” code and modifies the router configuration. Subtle indirection flaws are bad enough in your own code; in another’s code, they are nearly intractable.

3.4.4 Problematic Integration of “round-trip” Routers

The original Scout design documents discuss host requirements and describe video display units and web servers. The Scout data structures embed a “next” and “back” directionality, which permeates the system. In a host whose function is pure data-source (server) or data-sink (client) it is easy to visualize the “next” and “back” steps in a path. Multiple router interfaces (connection points to other routers) complicate matters somewhat, especially due to the anonymous nature of the interface references. At some later point, perhaps with the introduction of the TCPF router, “round-trip” paths were introduced into the scout architecture. These routers are essentially a bend in the processing path, so those messages that were traveling “up” towards the application layer are redirected back “down” towards the device layer.

While the Scout platform was an obvious platform for prototyping such work, the core Scout architecture concepts don’t map well onto U-turn paths. This is unfortunate, as the TCPF router and its excellent performance characteristics were the strongest selling point of the Scout architecture. Now that the round-trip router experiments (TCPF and ASP) have demonstrated their functionality and performance merits, Scout’s directionality assumptions and end-host-centric bias need to be reexamined. A new architecture paradigm is outside the scope of this document.

4 Nature of Problems

4.1 Debugging Issues

Cross-platform debugging is never easy. Embedded system development presents its own set of challenges, which the typical debugging environments either take for granted (endian-ness issues, executable delivery) or have matured beyond (bootstrap debugging, crash interception). Scout provides a framework for trace statements to be generated during run-time, and provides compile-time options for consistency checks on some types or message transactions. We did, however, need to augment these diagnostic aids with enhanced memory management routines, improved support for gdb on the Alpha platform, and various *ad hoc* mechanisms for tracking object corruption. Scout is a kernel, and kernel debugging is difficult.

The Alpha platform doesn’t have powerful hardware-based debug registers needed for some of the thornier bug tracing. One particular instability, which eventually proved to be related to a re-entrancy problem in the demultiplex path, only exhibited itself on the Alpha processor under heavy loads. The Intel Pentium processor has support for hardware debug registers that allow

read-write breakpoints, but the Acenic device associated with the heavy load was not currently running under the Intel port. Only after agonizing divide-and-conquer routines were sprinkled throughout the failing path were we able to isolate the errant pointer access. Once again, Scout is a kernel and kernel debugging is difficult. However, kernel debugging *can* be more effective with appropriate hardware assistance.

4.2 Device Driver Issues

In retrospect, we spent too much time focusing on the ATM driver support, which detracted from efforts on general high-performance-device support. As we had made a strategic decision midway through the project to direct our attention away from ATM signaling protocols and focus strictly on high-speed devices and IP, when the FORE HE622 driver port from Solaris to Scout dragged on and on we should have cut our losses and moved directly to Gigabit Ethernet. It was tempting to spend “just one more week” plugging away at the driver, particularly when we were able to generate ATM cells in the Scout platform (on transmit). We were never able to get the HE622 card to generate interrupts on receive. We were never able to find documented diagnostic registers in the devices to indicate whether frames were being received and dropped, received and buffered but the card wasn’t configured to generate interrupts directly, or whether the card was generating interrupts but we were not properly fielding them. In any event, we currently have a send-only ATM firewall, and a well functioning Gigabit Ethernet firewall. The saying goes: *For want of a nail, the Kingdom was lost*. In our case, *For want of an interrupt, the ATM firewall was lost*.

4.3 Workload Characterization Issues

It is difficult to define the “typical” firewall traffic profile⁷. Prior work at TIS and NAI Labs generated some “interesting” traffic tools and profiles, which proved somewhat useful in comparing ASP to commercial firewall products. However, traffic characterization is a multivariate space without clear definitions or standards. We eventually decided to sidestep the issue and take measurements of homogeneous traffic using the NAI Labs firewall performance testbed pictured below. We suggest that more work needs to be done to develop meaningful objective traffic profile characterization metrics.

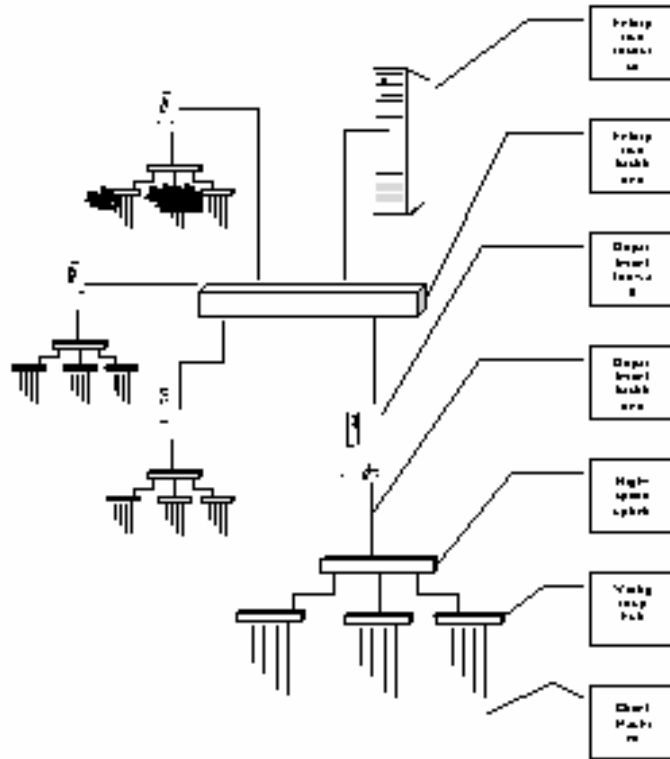


Figure 22 – NAILabs Firewall Performance Testbed

4.4 Community “Critical Mass”

As mentioned above, the Scout community hasn’t yet reached the critical mass required for self-sustaining growth. Aside from occasional source refreshes from the Arizona team, we pretty much had to “go it alone”. This is not to say that we share none of the blame; feedback to the community from the participants is vital to building the community spirit of all involved. As is often the case, project deadlines discouraged us from the extra amount of effort for frequent progress reports and source updates to the community. In addition, there is always the “just one more fix before I send this out” dilemma: often we were on the verge of sending a source release to the scout team but chose to delay the release until stability increased.

In hindsight, it would have been useful to include periodic “developer community” releases to the project timeline, to drive home the fact that users of open-source software need to take responsibility for sharing their improvements with the world. Frequent periodic releases, even with low stability, serve a useful function when there is an expectation of “brittleness” along with an expectation of frequent incremental improvements.

5 Details of Technical Work

5.1 HE622 Driver Details

The driver for the FORE HE622 OC-12c ATM card originated as a Solaris network driver. Porting this driver was done in two steps: (1) port to a non-Solaris platform using Gnu development tools (such as those used on Linux and Scout), and (2) port the now-compileable code to run in the Scout environment.

The Solaris device driver model provides a number of features not found in typical Linux or embedded environments, such as a configuration database and runtime-loadable modules. Other features in the Solaris device driver model are similar to, but not identical to, features that are found in a typical kernel environment and in the Scout architecture.

The HE622 device driver (also known as the “ForeThought” driver) provides a full ATM stack, including the UNI, CLIP, MPOA, and LANE. There is a full complement of ATM switching protocol support in the HE622 driver. The Scout port of the ForeThought driver was an iterative process, starting from the sample “sdapi” program included in the FORE source code distribution. SD-API is a minimal ATM frame transmit/receive application, which uses ATM Permanent Virtual Circuits (PVCs) to avoid the switching protocol overhead. The ASP ATM “router” actually wraps SD-API calls to generate and receive ATM cells, then performs simple SNAP LLC encapsulation for a rudimentary Ethernet-over-ATM protocol.

Only those features required by the SD-API protocol were actually ported to the Scout platform. As the FORE build environment dynamically detects the build tree at compile time, the irectory tree was modified to hide those components not included in the build, rather than make extensive changes to “Makefiles”.

As the FORE source was obtained under NDA from FORE Systems, we are not able to release the whole of the ATM driver. Instead, the ASP software distribution will include source patches, which must be applied to a specific version of the FORE software distribution in order to recreate the ASP environment.

5.2 ACENIC Driver Details

As it became apparent that the ATM driver support could not be finished in a timely manner, development efforts shifted to the creation of a Gigabit Ethernet testbed platform instead. As part of this effort, the Linux “AceNIC” driver was ported to both the Linux emulation framework within Scout and to the native Scout device driver framework. The native AceNIC driver is able to directly manipulate Scout message buffers, thereby avoiding two full-packet copies on receive and one on transmit. Accordingly, it is 30% more performant than its emulated Linux counterpart.

5.2.1 Compatibility

We currently support cards based on the Tigon I and Tigon II chipsets, including the Alteon AceNIC, the 3Com 3C985[B] and the NetGear GA620.

5.2.2 Chipset-Specific Settings

PCI bus devices are configured by the system BIOS at boot time, so no jumpers need to be set on the board. The system BIOS should be set to assign the PCI INTA signal to an otherwise unused system IRQ line. While it’s possible to share PCI interrupt lines, it negatively impacts performance. A 256-byte region of NIC memory can be controlled from the PCI configuration space. The NIC uniquely identifies itself and indicates to the host what it requires in terms of memory space, I/O space, bus latency, interrupt routing, and so on. The ASP/Scout kernel may modify these values during boot-time resource allocation.

The Tigon chipset can be tuned to wait, upon packet arrival, for additional packets to arrive before interrupting the host. This parameter is specified in microsecond clock ticks. Transmission

interrupts, i.e. notification that one or more packets have been successfully transmitted, may also be coalesced in a similar manner or specified to occur when a maximum number of transmit descriptors have been processed. Additionally, the ratio of the NIC's on-board memory for receive versus transmit buffering may be specified. On the 1MB NIC (generally, the only version available in recent years), approximately 800 kilobytes is available, and the default ratio is a 50/50 split.

5.2.3 Driver Operation

The Tigon presents six producer/consumer ring buffer interfaces to the host:

- Transmit descriptors (the “transmit ring”)
- Receive descriptors optimized for small buffers (the “mini ring”)
- Standard receive descriptors (the “standard ring”)
- Receive descriptors for jumbo frames, those with an MTU > 1514 and ≤ 9014 (the “jumbo ring”)
- Returning processed receive descriptors to the host (the “receive return ring”)
- Events (the “event ring”)

Events provide a means for the NIC and host to communicate status information “out of band” from receive and transmit processing. The NIC communicates firmware status, statistics, link state, and error notifications by way of the event interface.

The transmit ring and receive ring are populated with a fixed set of descriptors at driver initialization time. At this time the initial set of receive buffers is also allocated. During system operation, as receive buffers are consumed by the NIC, new receive buffers are allocated to replace those now queued for processing by the host IP stack. Transmit buffers are placed onto the transmit ring, and the host producer index is appropriately incremented, whenever the host has completed assembly of an outbound packet. It is worth noting here that the Tigon chipset supports automatic handling of misaligned buffers, so receive and transmit buffers suffer no alignment restrictions other than those imposed by the host architecture.

In accordance with the interrupt tuning parameters in effect, the Tigon chipset will interrupt host CPU processing at some point after any NIC receive producer, host transmit consumer, or NIC event ring producer index has been updated. In other words, the Tigon will interrupt the host when:

- One or more packets were consumed from a receive ring and subsequently placed on the receive return ring
- One or more queued transmit packets were sent
- One or more events were posted to the event ring

During interrupt processing, the device driver must refresh the receive source rings (the “mini”, “standard”, and “jumbo” rings) with new buffers as required. It must remove any buffers placed by the NIC onto the receive return ring, and queue those buffers for processing by the host IP stack. The driver also must remove those buffers on the transmit ring, if any, which the NIC indicates have been successfully sent. Finally, it must acknowledge any events placed on the event ring.

5.2.4 Porting Issues

The original Linux device driver's interrupt handler allocated memory out of the interrupt code path, in a background thread of control scheduled periodically as needed. The ASP/Scout

AceNIC driver does the same. In order to avoid starving the NIC under heavy load, however, it is necessary to force allocation of new buffers within the interrupt code path when a minimum threshold is reached. As the original Scout system services for allocating and freeing heap memory were not reentrant, we were forced to disable hardware interrupts during manipulation of critical data structures from these routines. This is an incomplete solution to a problem discovered late in the development of ASP. Scout requires a more robust memory management mechanism.

5.3 Hardware Acceleration Options

Owing to the original plan to incorporate ASP technology inside an ATM switch, but the loss of access to such switches, we continued through the life of the project to explore hardware-integration plans. Plausible integration targets do exist, particularly the Marconi SA400 device. The SA400 is a descendant of the NSA-developed CellBlock architecture: providing high-speed cell matching and forwarding. This Marconi product is an ATM firewall accelerator, which is implemented largely in Field Programmable Gate Arrays (FPGAs). We were able to obtain, under NDA, portions of the VHDL code which implements the cell matching and ATM/IP state following algorithms.

Using this VHDL code as a model for the kinds of operations available inside the SA400 device, we sketched paper designs which could eventually be used to implement first a hardware-assisted Network Address Translation device (Figure 23), and with slight refinements a hardware-assisted IPCUT engine (Figure 24).

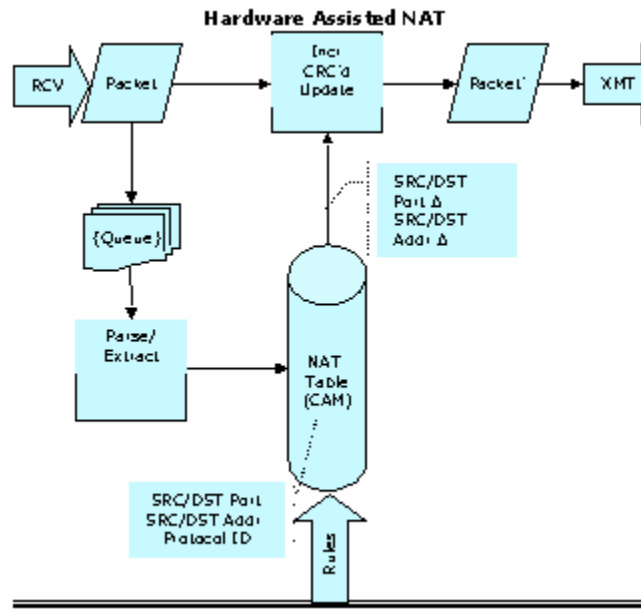


Figure 23 - Hardware Assisted NAT

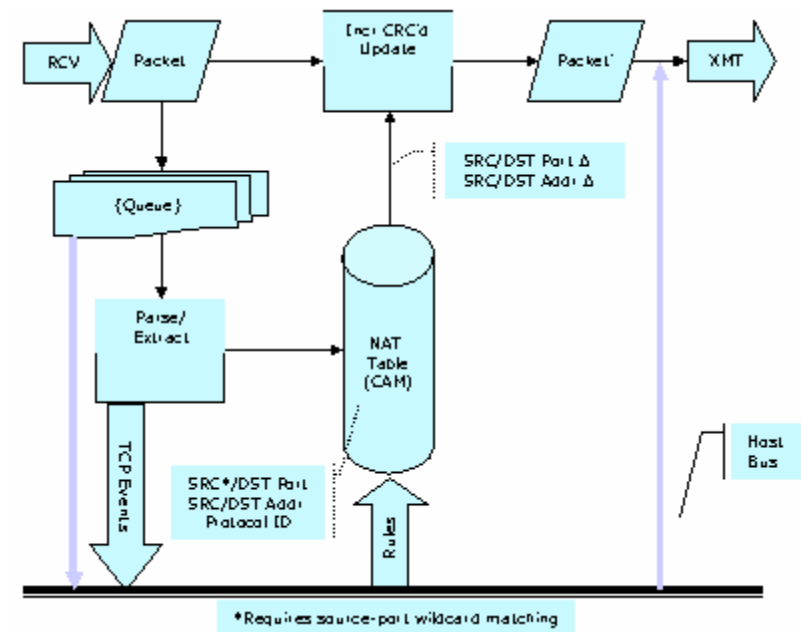


Figure 24 - Hardware Assisted IPCUT

We plan to keep these designs in mind for future high-confidence networking research.

References

-
- ¹ J.H. Hartman, A.B. Montz, D. Mosberger, S.W. O'Malley, L.L. Peterson, and T.A. Proebsting. Scout: A communication-oriented operating system. Technical Report TR 94-20, University of Arizona, Tucson, AZ, June 1994
- ² Oliver Spatscheck, Jorgen S. Hansen, John H. Hartman, and Larry L. Peterson. Optimizing TCP Forwarder Performance. Technical Report TR98-01, Dept. of Computer Science, University of Arizona, February 1998
- ³ Checkpoint Firewall-1, NAI Gauntlet, Cisco CIX, etc.
- ⁴ Circa 1998, <http://www.caida.org/outreach/resources/learn/trafficworkload/>
- ⁵ Hypertext Transfer Protocol (RFC 1945, RFC 2068)
- ⁶ MIME (RFC 2045, RFC 2046, RFC 2047)
- ⁷ B. A. Mah. "An Empirical Model of HTTP Network Traffic", Proc. InfoComm '97, April 1997
- ⁸ W. Willinger, V. Paxson, and M.S. Taqqu, "*Self-Similarity and Heavy-Tails: Structural Modeling of Network Traffic*," in A Practical Guide To Heavy Tails: Statistical Techniques and Applications, R.J. Adler, R.E. Feldman and M.S. Taqqu, editors. ISBN 0-8176-39519. Birkhauser, Boston, 1998.
- ⁹ Christos Faloutsos Michalis Faloutsos, Petros Faloutsos, "On power-law relationships of the internet topology," Proc. of ACM SIGCOMM, Aug. 1999, <http://citeseer.nj.nec.com/86873.html> , etc.
- ¹⁰ TELNET (RFC 854)
- ¹¹ Netperf (<http://www.netperf.org>)
- ¹² FTP (STD 9, RFC 959)
- ¹³ <http://ftp.arl.mil/ftp/pub/ttcp/>
- ¹⁴ <http://www.cs.princeton.edu/nsg/scout/>
- ¹⁵ Forrest J. Cavalier, III, "Some Implications of Bazaar Size", Third Draft. Aug 11, 1998, Copyright 1997-1998, [Mib Software](http://www.mibsoftware.com/bazdev/), <http://www.mibsoftware.com/bazdev/>
- ¹⁶ T. Proebsting and S. Watterson. Filter fusion. In Proc. Twenty-third ACM Symposium on Principles of Programming Languages, pages 119--130. ACM Press, 1996